Programming Strategies for Irregular Algorithms on the Emu Chick

Eric Hein[1], Srinivas Eswar[2], Abdurrahman Yasar[2], Jiajia Li[3], Jeffrey Young[2], Tom Conte[2], Ümit V. Çatalyürek[2], Rich Vuduc[2], Jason Riedy[2], and Bora Uçar[4]

[1]*Emu Technology*
[2]*Georgia Institute of Technology*
[3]*Pacific Northwest National Laboratory*
[4]*CNRS and LIP, École Normale Supérieure de Lyon*

Abstract

The Emu Chick prototype implements migratory memory-side processing in a novel hardware system. Rather than transferring large amounts of data across the system interconnect, the Emu Chick moves lightweight thread contexts to near-memory cores before the beginning of each remote memory read. Previous work has characterized the performance of the Chick prototype in terms of memory bandwidth and programming differences from more typical, non-migratory platforms, but there has not yet been an analysis of algorithms on this system.

This work evaluates irregular algorithms that could benefit from the lightweight, memory-side processing of the Chick and demonstrates techniques and optimization strategies for achieving performance in sparse matrix-vector multiply operation (SpMV), breadth-first search (BFS), and graph alignment across up to eight distributed nodes encompassing 64 nodelets in the Chick system. We also define and justify relative metrics to compare prototype FPGA-based hardware with established ASIC architectures. The Chick currently supports up to 68x scaling for graph alignment, 80 MTEPS for BFS on balanced graphs, and 50% of measured STREAM bandwidth for SpMV.

1 Introduction

Analyzing data stored in irregular data structures such as graphs and sparse matrices is challenging for traditional architectures due to limited data locality in associated algorithms and performance costs related to data movement. The Emu architecture [22] is designed specifically to address these data movement costs in a power-efficient hardware environment by using a cache-less system built around "nodelets" (see Figure 1) that execute lightweight threads. These threads migrate on remote data reads rather than pulling data through a traditional cache hierarchy. The key differentiators for the Emu architecture are the use of cache-less processing cores, a high-radix network

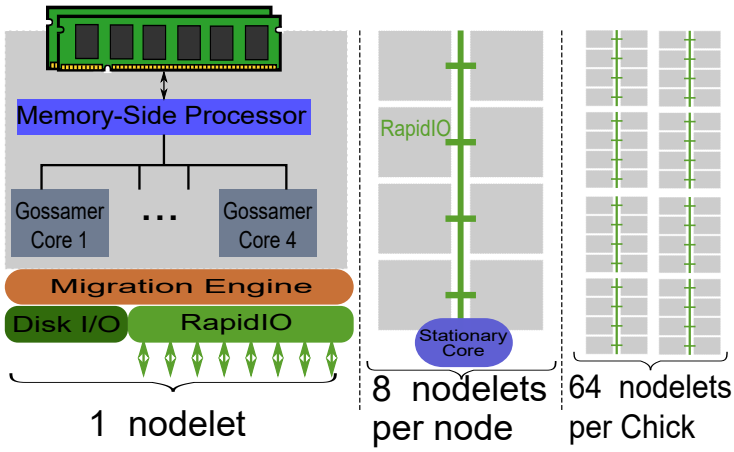


Figure 1: Emu architecture: The system consists of *stationary* processors for running the operating system and up to four *Gossamer* processors per nodelet tightly coupled to memory. The cache-less Gossamer processing cores are multi-threaded to both source sufficient memory references and also provide sufficient work with many outstanding references. The coupled memory's narrow interface ensures high utilization for accesses smaller than typical cache lines.

connecting distributed memory, and PGAS-based data placement and accesses. In short, the Emu architecture is designed to scale applications with poor data locality to supercomputing scale by more effectively utilizing available memory bandwidth and by dedicating limited power resources to networks and data movement rather than caches.

Previous work has investigated the initial Emu architecture design [22], algorithmic designs for merge and radix sorts on the Emu hardware [50], and baseline performance characteristics of the Emu Chick hardware [73, 11]. This investigation is focused on determining how irregular algorithms perform on the prototype Chick hardware and how we implement specific algorithms so that they can scale to a rack-scale Emu and beyond.

This study's specific demonstrations include:

- The first characterization of the Emu Chick hardware using irregular algorithms including sparse matrix vector multiply (SpMV), graph analytics (BFS), and graph alignment. We also discuss programming strategies for the Emu such as *replication* (SpMV), *remote writes to reduce migration* (BFS), and *data layout to reduce workload imbalance* (graph alignment) that can be used to increase parallel performance on the Emu.
- Multi-node Emu results for BFS scaling up to 80 MTEPS and 1.28 GB/s on a balanced graph as well as an initial comparison of Emu-optimized code versus a naive Cilk implementation on x86.
- Multi-node results for SpMV scaling up to 50% of measured peak bandwidth on the the Emu.
- Graph alignment results showing a 68× speedup when scaling from 1 to 256 threads on 8 nodelets with optimized data layout and comparison strategies.

Achieving these results produced a series of observations on programming the Emu platform. These observations, detailed in Section 6, can guide the Emu and future migratory thread systems.

2 The Emu Architecture

The Emu architecture focuses on improved random-access bandwidth scalability by migrating lightweight *Gossamer* threads, or "threadlets", to data and emphasizing fine-grained memory access. A general Emu system consists of the following processing elements, as illustrated in Figure 1:

- A common *stationary* processor runs the OS (Linux) and manages storage and network devices.
- *Nodelets* combine narrowly banked memory with highly multi-threaded, cacheless *Gossamer* cores to provide a memory-centric environment for migrating threads.

These elements are combined into nodes that are connected by a RapidIO fabric. The current generation of Emu systems include one stationary processor for each of the eight nodelets contained within a node. System-level storage is provided by SSDs. We talk more specifically about some of the prototype limitations of our Emu Chick in Section 4. More detailed descriptions of the Emu architecture are available [22], but this is a point in time description of the current implementation and its trade-offs.

For programmers, the Gossamer cores are transparent accelerators. The compiler infrastructure compiles the parallelized code for the Gossamer ISA, and the runtime infrastructure launches threads on the nodelets. Currently, one programs the Emu platform using Cilk [43], providing a path to running on the Emu for OpenMP programs whose translations to Cilk are straightforward. The current compiler supports the expression of task or fork-join parallelism through Cilk's `cilk_spawn` and `cilk_sync` constructs, with a future Cilk Plus (Cilk+) software release in progress that would include `cilk_for` (the nearly direct analogue of OpenMP's `parallel for`) as well as Cilk+ reducer objects. Many existing C and C++ OpenMP codes can translate almost directly to Cilk+.

A launched Gossamer thread only performs local reads. Any remote read triggers a migration, which will transfer the context of the reading thread to a processor local to the memory channel containing the data. Experience on high-latency thread migration systems like Charm++ identifies migration overhead as a critical factor even in highly regular scientific codes [1]. The Emu system minimizes thread migration overhead by limiting the size of a thread context, implementing the transfer efficiently in hardware, and integrating migration throughout the architecture. In particular, a Gossamer thread consists of 16 general-purpose registers, a program counter, a stack counter, and status information, for a total size of less than 200 bytes. The compiled executable is replicated across the cores to ensure that instruction access always is local. Limiting thread context size also reduces the cost of spawning new threads for dynamic data analysis workloads. Operating system requests are forwarded to the stationary control processors through the service queue.

The highly multi-threaded Gossamer cores read only local memory and do not have caches, avoiding cache coherency traffic. Additionally, "memory-side processors"

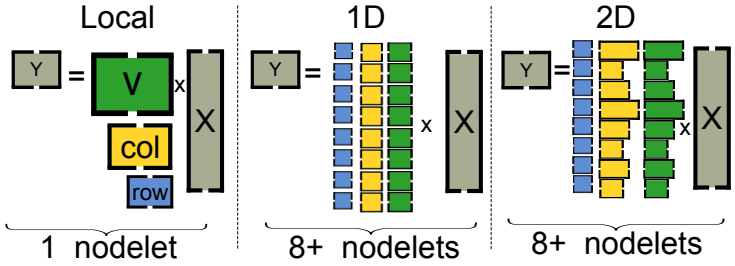


Figure 2: Distributed memory layouts for CSR SpMV (from [73]).

provide atomic read or write operations that can be used to access small amounts of data without triggering unnecessary thread migrations. A node's memory size is relatively large with standard DDR4 chips (64 GiB) but with multiple, Narrow-Channel DRAM (NCDRAM) memory channels (8 channels with 8 bit interfaces to the host using FIFO ordering). Each DIMM has a page size of 512B and a row size of 1024. The smaller bus means that each channel of NCDRAM has only 2GB/s of bandwidth, but the system makes up for this by having many more independent channels. Because of this, it can sustain more simultaneous fine-grained accesses than a traditional system with fewer channels and the same peak memory bandwidth.

3 Algorithms

We investigate programming strategies for three algorithms: 1) the standard (CSR) sparse matrix vector multiplication operation, 2) Graph500's breadth-first search (BFS) benchmark, and 3) graph alignment, computing a potential partial mapping of the vertices of two graphs. These three algorithms cover a variety of sparse, irregular computations: the ubiquitous sparse matrix vector multiplication, filtered sparse matrix sparse vector multiplication (in BFS), and a variant of the sparse matrix - sparse matrix multiplication (in computing the similarities of vertices). In the following subsections we discuss how we implement these algorithms on the Emu platform.

3.1 Sparse Matrix Vector Multiply (SpMV):

This algorithm computes the product of a sparse matrix A and a column vector X. Each element of the resulting column vector Y is computed as the dot product of X with a single row of A. The matrix A is stored in distributed memory using a compressed sparse row (CSR) layout consisting of 3 arrays - row offsets, column indices, and values. The row offset array is striped across all nodelets and encodes the length of each row. Every row's non-zero entries and column indices are allocated together and are present in the same nodelet giving rise to the jagged arrays `col` and V shown in Figure 2. X is replicated across each nodelet and the output Y is striped across all nodelets.

The 1D layout in Figure 2 stripes each array across the nodelets individually. The 2D layout stripes *blocks* of rows across nodelets but places the row data, adjacent columns and values, on the same nodelet (see [73] for details). In the 2D allocation case, no thread migrations occur when accessing elements in the same row.

A 1D striped layout incurs a migration for every element within a row to fetch the vector entry. The 2D layout is equivalent to that used in [59], but we consider the

Table 1: Notations used in BFS.

Symbol	Description
V	Vertex set
Q	Queue of vertices
P	Parent array
nP	New parent array
$\mathrm{Neig}(v)$	Neighbor vertices of v

impact of replicating data across the Chick.

Synthetic Laplacian matrix inputs are created corresponding to a d-dimensional k-point stencil on a grid of length n in each dimension. For the tested synthetic matrices, $d = 2$ and $k = 5$, resulting in a $n^2 \times n^2$ Laplacian with five diagonals. The Laplacian consists of the main diagonal, the 1st super and subdiagonals, and the nth super and subdiagonals. The upper and lower bandwidths of the synthetic matrices are n. The tested real world matrices are listed in Table 3.

3.2 Graph Analytics (Breadth First Search for Graph500)

A breadth-first search (BFS) begins at a single vertex of a graph. It explores all the neighbors of that vertex, then explores all the neighbors-of-neighbors, and continues in this fashion until all vertices connected to the initial vertex have been explored. Table 1 defines the notation used to refer to BFS data structures. Our in-memory graph layout is inspired by STINGER [23] so that computation can adapt to a changing environment[71]. Each vertex contains a pointer to a linked-list of edge blocks, each of which stores a fixed number of adjacent vertex IDs and a pointer to the next edge block. We use a striped array of pointers to distribute the vertex array across all nodelets in the system, such that vertex 0 is on nodelet 0, vertex 1 is on nodelet 1, and so on. We use STINGER rather than CSR to enable future work with streaming data and incremental algorithms[34], one of the primary targets of the Emu architecture. Note that breadth-first search is nearly equivalent to computing a filtered sparse matrix times sparse vector product[35].

To avoid the overhead of generic run-time memory allocation via `malloc`, each nodelet pre-allocates a local pool of edge blocks. A vertex can claim edge blocks from any pool, but it is desirable to string together edge blocks from the same pool to avoid thread migrations during edge list traversal. When the local pool is exhausted, the edge block allocator automatically moves to the pool on the next nodelet.

Kernel 1 of the Graph500 benchmark involves constructing a graph data structure from a list of edges. In our implementation the list of edges is loaded from disk into memory on nodelet 0. Currently I/O is limited on the prototype Emu Chick, and loading indirectly assists in evaluating the rest of the architecture. We sort the list by the low bits of the source vertex ID to group together edges that will be on the same nodelet, then spawn threads to scatter the list across all the nodelets. Once the list has been scattered, each nodelet spawns more threads locally to insert each edge into the graph, allocating edge blocks from the local pool.

Our initial implementation of BFS (Algorithm 1) was a direct port of the STINGER code. Each vertex iterates through each of its neighbors and tries to set itself as the

Algorithm 1: BFS algorithm using migrating threads.

```
P[v] ← −1, for ∀v ∈ V
Q.push(root)
while Q is not empty do
    for s ∈ Q do in parallel
        for d ∈ Neig(s) do in parallel
            ▷ Thread migrates reading P[d]
            if P[d] = −1 then
                if compare_and_swap(P[d], -1, s) then
                    Q.push(d)
    Q.slide_window()
```

parent of that vertex using an atomic compare-and-swap operation. If the operation is successful, the neighbor vertex is added to the queue to be explored along with the next frontier.

On Emu, the parent array is striped across nodelets in the same way as the vertex array. Each nodelet contains a local queue so that threads can push vertices into the queue without migrating. At the beginning of each frontier, threads are spawned at each nodelet to explore the local queues. Thread migrations do occur whenever a thread attempts to claim a vertex that is located on a remote nodelet. In the common case, a thread reads an edge, migrates to the nodelet that owns the destination vertex, executes a compare-and-swap on the parent array, pushes into the local queue, and then migrates back to read the next edge. If the destination vertex happens to be local, no migration will occur when processing that edge.

Algorithm 2: BFS algorithm using remote writes

```
P[v] ← −1, for ∀v ∈ V
nP[v] ← −1, for ∀v ∈ V
Q.push(root)
while Q is not empty do
    for s ∈ Q do in parallel
        for d ∈ Neig(s) do in parallel
            ▷ Thread issues remote write to nP[d]
            nP[d] ← s
    cilk_sync
    for v ∈ V do in parallel
        if P[v] = −1 then
            if nP[v] ≠ −1 then
                P[v] ← nP[v]
                Q.push(v)
    Q.slide_window()
```

An alternative BFS implementation (Algorithm 2) exploits the capability of the Emu system to efficiently perform remote writes. A copy of the parent array (*nP*) holds intermediate state during each frontier. Rather than migrating to the nodelet that contains the destination vertex, we perform a remote write on the *nP* array. The remote

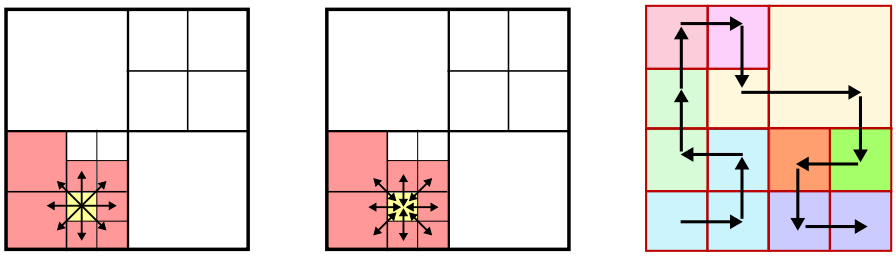
write packet can travel through the network and complete asynchronously while the thread that created it continues to traverse the edge list. Remote writes attempting to claim the same vertex are serialized in the memory front end of the remote nodelet. Rather than attempting to synchronize these writes, we simply allow later writes to overwrite earlier ones. After all the remote writes have completed, we scan through the *nP* array looking for vertices that did not have a parent at the beginning of this frontier ($P[v] = -1$) but were assigned a parent in this iteration ($nP[v] \neq -1$). When such a vertex is found, it is added to the local queue, and the new parent value $nP[v]$ is copied into the parent array at $P[v]$. This is similar to direction-optimizing BFS [10] and may be able to adopt its early termination optimizations.

3.3 GSANA: Parallel Similarity Computation

Integrating data from heterogeneous sources is often modeled as merging graphs. Given two or more compatible, but not necessarily isomorphic graphs, the first step is to identify a *graph alignment*, where a potentially partial mapping of the vertices of the two graphs is computed. In this work, we investigate the parallelization of GSANA [68], which is a recent graph aligner that uses the global structure of the graphs to significantly reduce the problem space and align large graphs with a minimal loss of information. The proposed techniques are highly flexible, and they can be used to achieve higher recall while being orders of magnitude faster than the current state of the art [68].

Briefly, GSANA first reduces the problem space, then runs pairwise similarity computation between two graphs. Although the problem space can be reduced significantly, the pairwise similarity computation step remains to be the most expensive part (more than 90% of the total execution time). While GSANA has an embarrassingly parallelizable nature for similarity computations, its parallelization is not straightforward. This is because of the fact that GSANA's similarity function is composed of multiple components, with some only depending on graph structure and others depending also on the additional metadata (types and attributes). All of these components compare vertices from two graphs and/or their neighborhood. Hence, the similarity computation step has a highly irregular data access pattern. To reduce this irregularity, we store the metadata of a vertex's neighborhood in sorted arrays. While arranging metadata helps to decrease irregularity, data access remains a problem because of the skewed nature of real-world graphs. Similarity computations require accessing different portions of the graph simultaneously. In [69] authors provide parallelization strategies for different stages of GSANA. However, because of the differences in the architecture and the parallelization framework, the earlier techniques cannot be applied to EMU Chick in a straightforward manner. Hence, in this work, we investigate two parallelization strategies for similarity computations and also two graph layout strategies on Emu Chick.

GSANA places vertices into a $2D$ plane using a graph's global structure information. The intuition is that similar vertices should also have similar structural properties, and they should be placed closely on the $2D$ plane. When all vertices are placed, GSANA partitions space into buckets in a quad-tree like fashion. Then, a task for similarity computation becomes the pairwise comparison of the vertices in a bucket with vertices



(a) All (b) Pair (c) Layout

Figure 3: GSANA: Task definition & bucket and vertex partition among the nodelets respecting the Hilbert-curve order.

Table 2: Notations used in GSANA.

Symbol	Description
V_1, V_2	Vertex sets
E_1, E_2	Edge sets
QT_1, QT_2	Quad-trees of the graphs
$QT_i.Neig(B)$	Neighboring buckets of B in QT_i
$\sigma(u, v)$	Similarity score for $u \in V_1$ and $v \in V_2$
$N(u)$	Adjacency list of $u \in V_i$
$A(u)$	Vertex attribute of $u \in V_i$
$RW(f(\cdot))$	Number of required memory Reads & Writes to execute given function, $f(\cdot)$

in the neighboring buckets. For example, in Figures 3a-3b the vertices in the yellow colored bucket are compared with vertices in the yellow and red colored buckets. We investigate two parallel similarity computation schemes and two vertex layout schemes. Refer to Table 2 for the definition of notations used in these algorithm listings.

Algorithm 3: PARALLELSIM(QT_1, QT_2, k, σ)

$P[v] \leftarrow \emptyset$, for $\forall v \in V_2$
for each *non-empty* $B \in QT_2$ **do**
  `cilk_spawn` COMPSIM($B, QT_1.Neig(B), P, \sigma$)
`cilk_sync`
return P

3.3.1 Similarity computation schemes.

In the *All Comparison* scheme, Alg. 3 first spawns a thread for each non-empty bucket of $B \in QT_2$ where COMPSIM is instantiated with COMPSIMALL shown in Alg. 4. This function computes the similarity scores for each vertex $v \in B$ with vertex $u \in B'$ where $B' \in QT_1.Neig(B)$. Afterwards, the top k similar vertices are identified and stored in $P[v]$. This technique is illustrated in Figure 3a.

In the *Pair Comparison* scheme, Alg. 3 first spawns a thread for each non-empty bucket of $B \in QT_2$ where COMPSIM is instantiated with COMPSIMPAIR shown in Alg. 5. Then, for each $\langle B, B' \rangle$ pair where $B \in QT_2$ and $B' \in QT_1.Neig(B)$, COMPSIM-

Algorithm 4: compSimAll(B, N_B, P, σ)

```
▷ For each vertex keep a priority list with top k elements.
for each v ∈ B do
    for each B' ∈ N_B do
        for each u ∈ B' do
            P[v].insert(u)                          ▷ Only keeps top k
return P
```

PairAux is spawned. Next we compute pairwise similarity scores of vertices between these bucket pairs and return intermediate similarity scores (see Alg. 5). Finally, we merge these intermediate results in Alg. 5. This scheme spawns much more threads than the previous one. This technique is illustrated in Figure 3b.

In ALL comparison scheme, the number of threads is limited by the number of buckets. Therefore achievable scalability is limited. Furthermore, coarse grain decomposition of the tasks may lead to high load balance. Sorting tasks based on their loads in a non-increasing order can be a possible optimization/heuristic for reducing imbalance.

The PAIR comparison scheme reduces the load imbalance by compromising with additional synchronization cost that arises during the insertion in Alg. 4. Task list is shuffled to decrease the possibility of concurrent update requests to a vertex's queue.

Note that while *ALL* is akin to vertex-centric based partitioning, *PAIR* is akin to edge-based partitioning. The vertices and edges here refer to the task graph.

3.3.2 Vertex layouts.

In the *Block partitioned* (BLK) layout, the vertices are partitioned among the nodelets based on their *ID*s, independent from their placement in the 2D plane. The buckets are also partitioned among the nodelets independently. That is, each nodelet stores an equal number of vertices and buckets. A vertex's metadata is also stored in the same nodelet of corresponding vertex. With the two computational schemes, vertices in the same bucket may be in different nodelets, leading to many thread migrations. In the *Hilbert-curve based* (HCB) layout (shown in Fig. 3c), the vertices and buckets are partitioned among nodelets based on their Hilbert orders. To achieve this, after all vertices are inserted in the quad-tree, we sort buckets based on their Hilbert orders. Then, we relabel every vertex in a bucket according to bucket's rank (i.e., vertices in the first bucket, B, have labels starting from 0 to $|B| - 1$). In this layout every vertex is placed in the same nodelet with its bucket. As with *BLK*, a vertex's metadata is also stored in the same nodelet of the corresponding vertex. Here, all vertices in the same bucket are in the same nodelet, and hence there is in general less migration. While *BLK* may lead to a better workload balance (equal number of similarity computations per nodelet), *HCB* may lead to a workload imbalance, if two buckets with high number of neighbors are placed into the same nodelet.

Algorithm 5: compSimPair(B, N_B, P, σ)

$p_{B'} \leftarrow \emptyset$, for $\forall B' \in N_B$
for each $B' \in N_B$ **do**
  `cilk_spawn` compSimPairAux($B, B', p_{B'}, \sigma$)
`cilk_sync`
$P \leftarrow$ merge($p_{B' \in N_B}$)
return P

def compSimPairAux(B, B', P, σ):
▷ For each vertex keep a priority list with top k elements.
for each $v \in B$ **do**
  for each $u \in B'$ **do**
    $P[v].insert(u)$ ▷ Only keeps top k
return P

4 Experimental Setup

4.1 Emu Chick Prototype

The Emu Chick prototype is still in active development. The current hardware iteration uses an Arria 10 FPGA on each node card to implement the Gossamer cores, the migration engine, and the stationary cores. Several aspects of the system are scaled down in the current prototype with respect to the next-generation system which will use larger and faster FPGAs to implement computation and thread migration. The current Emu Chick prototype has the following features and limitations:

- Our system has one Gossamer Core (GC) per nodelet with a concurrent max of 64 threadlets. The next-generation system will have four GC's per nodelet, supporting 256 threadlets per nodelet.
- Our GC's are clocked at 175MHz rather than the planned 300MHz in the next-generation Emu system.
- The Emu's DDR4 DRAM modules are clocked at 1600MHz rather than the full 2133MHz. Each node has a peak theoretical bandwidth of 12.8 GB/s.
- CPU comparisons are made on a four-socket, 2.2 GHz Xeon E7-4850 v3 (Haswell) machine with 2 TiB of DDR4 with memory clocked at 1333 MHz (although it is rated for 2133 MHz). Each socket has a peak theoretical bandwidth of 42.7 GB/s.
- The current Emu software version provides support for C++ but does not yet include functionality to translate Cilk Plus features like `cilk_for` or Cilk `reducers`. All benchmarks currently use `cilk_spawn` directly, which also allows more control over spawning strategies.

4.2 Experiment Configurations

All experiments are run using Emu's 18.09 compiler and simulator toolchain, and the Emu Chick system is running NCDIMM firmware version 2.5.1, system controller software version 3.1.0, and each stationary core is running the 2.2.3 version of software. We present results for several configurations of the Emu system:

- Emu Chick single-node **(SN)**: one node; 8 nodelets
- Emu Chick multi-node **(MN)**: 8 nodes; 64 nodelets
- Simulator results are *excluded from this study* as previous work [73] has shown

simulated scaling numbers for SpMV and STREAM on future Emu systems. We prioritize multi-node results on hardware.

Application inputs are selected from the following sources:

- The SpMV experiments use synthetic Laplacian matrices, and real-world inputs are selected from the SuiteSparse sparse matrix collection [38]. Each Laplacian consists of a five-point stencil which is a pentadiagonal matrix.
- BFS uses RMAT graphs as specified by Graph500 [6] and uniform random (Erdös-Renyi) graphs [72], scale 15 through 21, from the generator in the STINGER codebase[1].
- GSANA uses DBLP [54] graphs from years 2015 and 2017 that have been created previously [68]. Detailed description of these graphs is provided in Section 5.3.

4.3 Choosing Performance Metrics

The Emu Chick is essentially a memory to memory architecture, so we primarily present results in terms of memory bandwidth and effective bandwidth utilization. But comparing a new and novel processor architecture (Emu) built on FPGAs to a well-established and optimized architecture built on ASICs (Haswell) is difficult. Measuring bandwidth on the Haswell with the STREAM benchmark[47] achieves much more of the theoretical peak memory bandwidth. The Emu Chick, however, implements a full processor on an FPGA and cannot take advantage of deeply pipelined logic that gives boosts to pure-FPGA accelerators, thus cannot achieve much of the theoretical hardware peak. If we compare bandwidths against the DRAM peaks, prototype novel architectures like the Chick almost never appear competitive. Comparing against measured peak bandwidth may provide an overly optimistic view of the prototype hardware.

We have chosen to primarily consider percentage of measured peak bandwidth given an idealized problem model, but also report the raw bandwidth results. For integer SpMV and BFS, the natural measures of IOPS (integer operations per second) and TEPS (traversed edges per second) are proportional to the idealized effective bandwidth.

Our more recent tests have shown that the Emu hardware can achieve up to 1.6 GB/s per node and 12.8 GB/s on 8 nodes for the STREAM benchmark, which is used as the measured peak memory bandwidth number. This increase in STREAM BW from previous work [73] is primarily due to clock rate increases and bug fixes to improve system stability. Meanwhile, our four-socket, 2.2GHz Haswell with 1333 MHz memory achieves 100 GB/s, or 25 GB/s per NUMA domain. So the Emu FPGA-emulated processors achieve 11.7% of the theoretical peak, while the ASIC Haswell processors achieve 58.6%. Note that we run with NUMA interleaving enabled, so many accesses cross the slower QPI links. This provides the best Haswell performance for our pointer chasing benchmark[73]. Disabling NUMA page interleaving brings the Haswell STREAM performance to 160 GB/s, which is 94% of the theoretical peak.

[1] `https://github.com/stingergraph/stinger/commit/149d5b562cb8685036517741bd6a91d62cb89631`

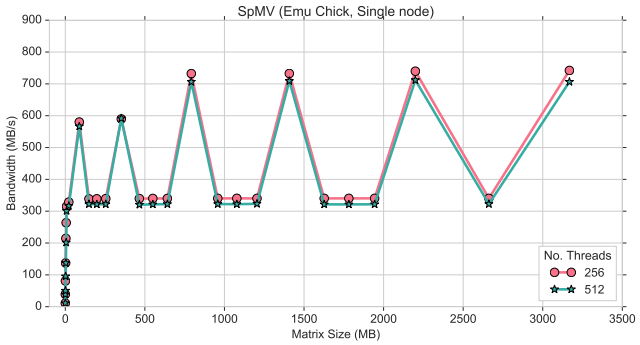


Figure 4: SpMV Laplacian Stencil Bandwidth, No Replication (8 nodelets).

5 Results

5.1 SpMV - to replicate or not, that is the question

We first look at the effects of replication on the Emu, that is whether replicating the vector x in Fig. 2 provides a substantial benefit when compared to striping x across nodelets in the "no replication" case.

Effective Bandwidth is the primary metric measured in our experiments. It is calculated as the minimum number of bytes needed to complete the computation. On cache-based architectures this is equivalent to the compulsory misses. For SpMV it is approximated by,

$$BW = \frac{\texttt{sizeof}(A) + \texttt{sizeof}(x) + \texttt{sizeof}(y)}{\text{time}}$$

The numerator is a trivial lower-bound on data moved, since it counts only one load of A (which enjoys no reuse) and one load each of the two vectors, x and y (assuming maximal reuse). The motivation for ignoring multiple loads of x or y is that ideally on a cache-based architecture with a "well-ordered" matrix, the vectors are cached and the computation is bandwidth-limited by the time to read A.

Figure 4 shows that the choice of grain size, or iterations / work assigned to a thread, can dramatically affect performance for the non-replicated test case. The unit of work here is the number of rows assigned to each thread. A fixed grain size of 16, while competitive for smaller graphs, does not scale well to the entire node. For small grain sizes, too many threads are created with little work per thread, resulting in slowdown due to thread creation overhead. A dynamic grain size calculation is preferred to keep the maximum number of threads in flight, as can be seen with the peak bandwidth achieved with 256 and 512 threads for a single node. Spikes in performance occur whenever the matrix's dimension is perfectly divisible by the threads per nodelet. This suggests that the spikes occur whenever work is perfectly load balanced across threads within a nodelet. Since we are using 32 and 64 threads per nodelet in Figure 4 this is seen for Laplacian Stencil sizes (n) of 1000, 2000, 3000, 4000, 5000 and 6000.

Figure 5 shows the effects of replication in SpMV. Interestingly, for the largest matrix size both Figures 4 and 5 have similar bandwidths, which indicates good scaling for larger data sizes without replication at the potential cost of thread migration hotspots

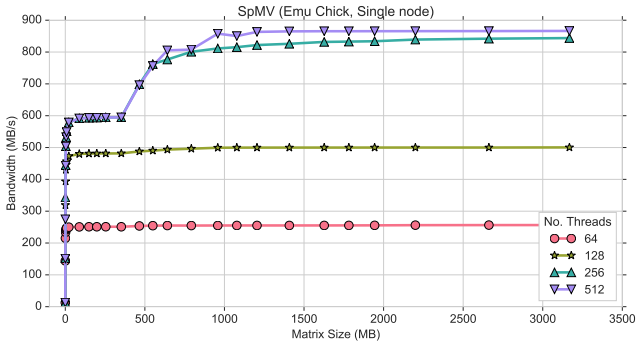


Figure 5: SpMV Laplacian Stencil Bandwidth, Replication (8 nodelets).

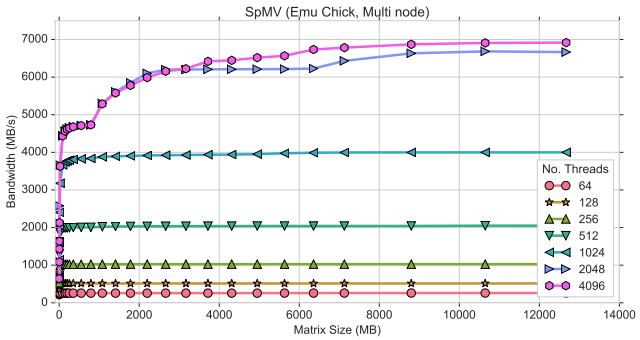


Figure 6: SpMV Laplacian Replicated - multinode (64 nodelets).

on certain nodelets. Without replication, we are guaranteed at least 2 migrations per row of the Laplacian Stencil due to the presence of the 1st super and sub diagonals (Section 3.1). However, we note that using replication leads to much more regular scaling curves across different numbers of threads and grain sizes.

Figure 6 shows scaling of multi-node (64 nodelets) using replication and different numbers of threads. The best run of SpMV achieves 6.11 GB/s with 4096 threads, which is 50.8% of our updated STREAM bandwidth number. However, it should also be noted from this figure that the best scalability for all sizes (including smaller inputs) is achieved using 2048 threads.

Table 3 shows the multi-node (run with 2048 threads) bandwidth in MB/s for real-world graphs along with their average and maximum degree (non-zero per row) values. The rows are sorted by maximum degree and if we exclude the graphs with large maximum degree (≥ 600) we see similar bandwidths. Most graphs showed bandwidths in excess of 600 MB/s and many were comparable to that of the synthetic Laplacians which are very well structured. This behavior is in contrast to a cache based system where we expect performance to increase with increasing degree. The Emu hardware demonstrates good performance independent of the structure of the graph, even ones with high-degree vertices. However this performance depends on replicating the vector X on each nodelet, which might not be possible at larger scales.

For the high maximum degree graphs (*Stanford*, *ins2*) we attribute the poor perfor-

Table 3: SpMV multinode bandwidths (in MB/s) for real world graphs [38] along with matrix dimension, number of non-zeros (NNZ), and the average and maximum row degrees. Run with 4K threads

Matrix	Rows	NNZ	Avg Deg	Max Deg	BW
mc2depi	526K	2.1M	3.99	4	3870.31
ecology1	1.0M	5.0M	5.00	5	4425.61
amazon03	401K	3.2M	7.99	10	4494.79
Delor295	296K	2.4M	8.12	11	4492.47
roadNet-	1.39M	3.84M	2.76	12	3811.57
mac_econ	206K	1.27M	6.17	44	3735.54
cop20k_A	121K	2.62M	21.65	81	4520.05
watson_2	352K	1.85M	5.25	93	3486.30
ca2010	710K	3.49M	4.91	141	4075.97
poisson3	86K	2.37M	27.74	145	4031.20
gyro_k	17K	1.02M	58.82	360	2446.36
vsp_fina	140K	1.1M	7.90	669	1335.59
Stanford	282K	2.31M	8.20	38606	287.82
ins2	309K	2.75M	8.89	309412	43.91

mance to load imbalance. Some of the rows in these graphs have a very high number of non zeros. Since we only partition work at the row level, a single thread will need to process these large rows and this load imbalance results in slow running times. Current hardware limitations prevent exploring mixing parallelism across and within matrix rows [57] leaving that level of performance benefit to future work.

5.2 Graph500 - Migrating versus Remote Writes

Figure 7 compares the migrating threads and remote write BFS implementations for a "streaming" or unordered BFS implementation. With the migrating threads algorithm, each thread will generally incur one migration per edge traversed, with a low amount of work between migrations. The threads are blocked while migrating, and do not make forward progress until they can resume execution on the remote nodelet. In contrast, the remote writes algorithm allows each thread to fire off many remote, non-blocking writes, which improves the throughput of the system due to the smaller size of remote write packets.

The effective bandwidth for BFS on a graph with a given scale and an edge factor of 16 is as follows:

$$BW = \frac{16 \times 2^{\text{scale}} \times 2 \times 8}{\text{time}} = TEPS \times 2 \times 8.$$

This does not include bandwidth for flags or other state data structures and so is a lower bound as discussed in Section 5.1.

Our initial graph engine implementation does not attempt to evenly partition the graph across the nodelets in the system. The neighbor list of each vertex is co-located with the vertex on a single nodelet. RMAT graphs specified by Graph500 have highly skewed degree distributions, leading to uneven work distribution on the Emu. Figure 8 shows that BFS with balanced Erdös-Rényi graphs instead leads a higher performance

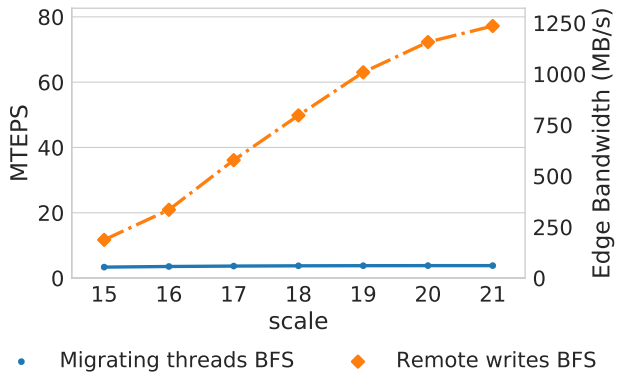


Figure 7: Comparison of remote writes versus migrating BFS on a multi-node Chick system for balanced (Erdös-Rényi) graphs. Marking members of the frontier with remote writes is more efficient than moving entire thread contexts back and forth between the edge list and the parent array.

of 18 MTEPS (288 MB/s) versus 4 MTEPS (64 MB/s) for the RMAT graph. We were unable to collect BFS results for RMAT graphs on the multi-node Emu system due to a hardware bug that currently causes the algorithm to deadlock. Future work will enhance the graph construction algorithm to create a better partition for power-law graphs.

Figure 9 plots results for four configurations of BFS running with balanced graphs: Emu single- and multi-node and two BFS results from the Haswell system. The performance of a single node of the Emu Chick saturates at 18 MTEPS while the full multi-node configuration reaches 80 MTEPS on a scale 21 graph, with an equivalent bandwidth utilization of 1280 MB/s. On the Haswell platform, the MEATBEE (backported Emu Cilk) implementation reaches a peak of 105 MTEPS, outperforming the STINGER (naive Cilk) implementation of BFS at 88 MTEPS, likely due to the reduction of atomic operations as discussed in Section 3.2.

5.3 GSANA Graph Alignment - Data Layout

For our tests, we use DBLP [54] graphs from years 2015 and 2017 that have been created previously [68]. This pair of graphs is called DBLP (0), and they have nearly $48K$, $59K$ vertices and $453K$, $656K$ edges respectively. These graphs are used in the experiments shown in Fig. 10. For the experiments shown in Fig. 11, we filter some vertices and their edges from the two graphs in DBLP (0), resulting in seven different graph pairs for alignment. The properties of these seven pairs are shown in Table 4.

We present similarity computation results for the Emu hardware on different sized graphs and execution schemes which are defined/named by combining the layout with the similarity computation. For instance, *BLK-ALL* refers to the case where we use the block partitioned vertex layout and run *ALL* parallel similarity computation. **Bandwidth** is measured for GSANA by the formula:

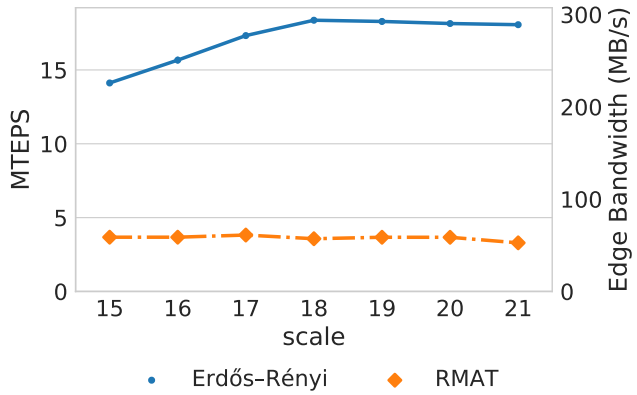


Figure 8: Compares the performance of BFS on a single-node system between balanced (Erdös-Rényi) graphs and unbalanced (RMAT) graphs running on a single node of the Emu Chick. Unbalanced graphs lead to an uneven work distribution and low performance.

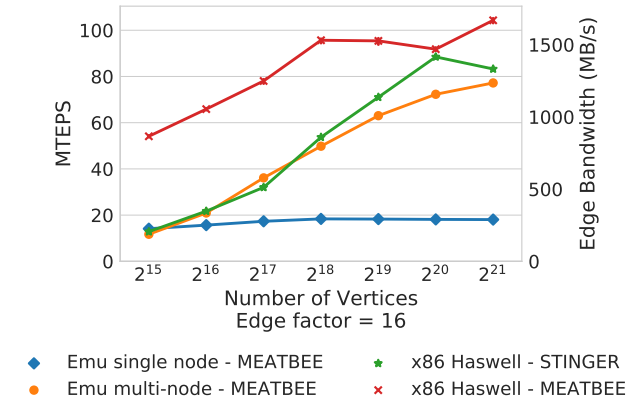


Figure 9: Comparison of BFS performance between the Emu Chick and the Haswell server described in Section 4. Two implementations were tested on the Haswell System, one from STINGER and the other from MEATBEE.

Table 4: Generated graphs for alignment. $K = 1024$; $|T|$: number of tasks; $|B|$: bucket size.

Graphs:	512	1024	2048	4096	8192	16384	32768
$\|V_1\|, \|V_2\|$	$0.5K$	K	$2K$	$4K$	$8K$	$16K$	$32K$
$\|E_1\|$	$1.3K$	$4.4K$	$14K$	$35K$	$88K$	$186K$	$385K$
$\|E_2\|$	$1K$	$3K$	$15K$	$30K$	$69K$	$147K$	$310K$
$\|T\|$	44	85	77	163	187	267	276
$\|B\|$	32	32	64	64	128	128	256

$$BW = \sum_{\forall B \in QT_1} \sum_{\substack{\forall B' \in \\ QT_2.Neig(B)}} \frac{\left(|B| + |B||B'| + \sum_{\forall u \in B} \sum_{\forall v \in B'} RW(\sigma(u,v))\right) \times \texttt{sizeof}(u)}{\texttt{time}}$$

Note that GSANA spends more than 90% of the total execution time for the similarity computation on an Intel Haswell CPU. This work focuses on the the similarity computation stage. We follow an offline approach; vertex layouts are created and then written into binary files on a Haswell-based machine. The layout files are read as inputs on the Emu platform. This pre-processing time takes less than a second on a Haswell CPU. Our Emu execution timings do not include the reading these input files.

In a task, pairwise vertex similarities are computed between the vertices in a bucket $B \in QT_1$ and the vertices in a bucket $B' \in QT_2.Neig(B)$. Therefore in each task, every vertex $u \in B$ is read once and every vertex $v \in B'$ is read $|B|$ times. Additional read and write cost comes from the similarity function $\sigma(u,v)$ that is called for every vertex pair u, v with $u \in B$ and $v \in B'$. Hence, the total data movement can be gathered by aggregating the size of the bucket reads and the size of the number of reads and writes required by the similarity function. Bandwidth is the ratio between the total data movement over the execution time. We adopted the following similarity metrics from GSANA [68]: degree (Δ), vertex type (τ), adjacent vertex type (τ_V), adjacent edge type (τ_E), and vertex attribute (C_V). Since the similarity function consists of four different similarity metrics, we can define the required number of reads and writes of the similarity function as $RW(\sigma(u,v)) = RW(\tau(u,v)) + RW(\delta(u,v)) + RW(\tau_V(u,v)) + RW(\tau_E(u,v)) + RW(C_V(u,v))$. In this equation, the degree (Δ) and the type (τ) similarity functions require one memory read for each vertex and then one read and update for the similarity value. Therefore, $RW(\tau(u,v)) = RW(\Delta(u,v)) = 4$. The adjacent vertex ($\tau_V$) and the edge ($\tau_E$) type similarity functions require reading all adjacent edges of the two vertices and one read and update for the similarity value. Therefore, $RW(\tau_V(u,v)) = RW(\tau_E(u,v)) = |N(u)| + |N(v)| + 2$. The vertex attribute similarity function (C_V) requires reading attributes of the two vertices and one read and update for the similarity value. Therefore, $RW(C_V(u,v)) = |A(u)| + |A(v)| + 2$.

The last three similarity metrics from GSANA [68] require comparing the neighborhood of two vertices, which causes a significant number of thread migrations if the two vertices appear in different nodelets. Therefore, these metrics are good candidates to test the capabilities of the current hardware.

Figure 10 displays the bandwidth results of the similarity computation schemes for increasing numbers of threads, in different execution schemes. In these experiments, we only present results of the *PAIR* computation scheme with the largest number of threads. Since the *PAIR* scheme does many unpredictable recursive spawns, controlling the number of threads for this scheme is very hard and not accurate. Therefore, for increasing number of threads, we only consider *ALL* with *BLK* and *HCB* vertex layouts. We observe that in the *BLK* layout, our final speedup is $43\times$ using *ALL* and $52\times$ using *PAIR*. In the *HCB* layout, our final speedup is $49\times$ using *ALL* and $68\times$ using *PAIR*. As can be seen in Fig. 10, when we increase the number of threads from 128 to 256, the

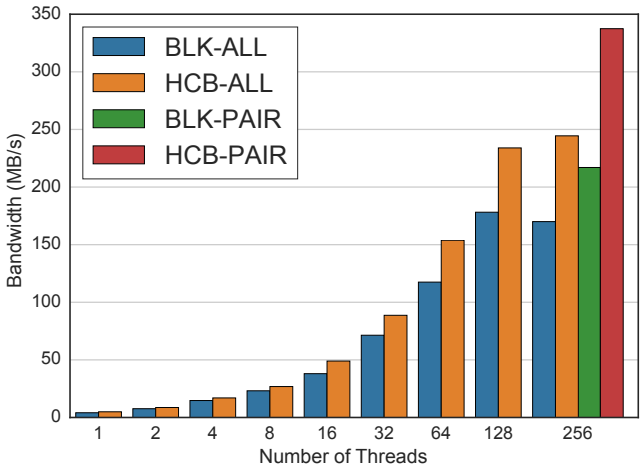


Figure 10: GSANA, Bandwidth vs. Threads for *ALL* (rightmost bars represent *PAIR* results), run on HW (8 nodelets).

bandwidth decreases by 4% in *BLK-ALL* scheme, because the coarse grained nature of *ALL* cannot give better workload balance and thread migrations hurt the performance.

Figure 11 presents results for all graphs in different execution schemes. We observe that the *HCB* vertex layout improves the execution time by 10-to-36% in all datasets by decreasing the number of thread migrations. As expected, this improvement increases with the graph size. This improvement in a x86 architecture is reported as 10% in [69]. Second, we see that the *PAIR* computation scheme enjoys improvements with both vertex layouts, because it has a finer grained task parallelism and hence better workload distribution.

Figure 12 displays strong scaling results for BLK and HCB vertex layouts with the *ALL* scheme on single-node and multi-node setups for the DBLP graph with 2048 vertices. Here, the strong scaling is given with respect to the single thread execution time of the BLK layout on the multi-node setup. On the multi-node setup, hardware crashed for GSANA when 128 threads were used. We observe from this figure that multi-node setup is slower than the single node setup—multi-node execution times are about 25% to 30% slower than the single-node execution times. This is so as the inter-node migrations are much more expensive. The proposed layout and computational schemes help to improve efficiency of the algorithms on both multi-node and single-node experiments. HCB layout improves ALL layout about 12% to 3%.

Final observations: We observe that the finer granularity of tasks in *PAIR* and locality aware vertex layout with *HCB* give an important improvement in terms of the bandwidth (i.e., execution time). However, because of recursive spawns *PAIR* may cause too many unpredictable thread migrations if the data layout is random. Additionally, although *HCB* helps to reduce the number of thread migrations significantly, this layout may create hotspots if it puts many neighboring buckets into the same nodelet. Our approach of balancing the number of edges per nodelet tries to alleviate these issues.

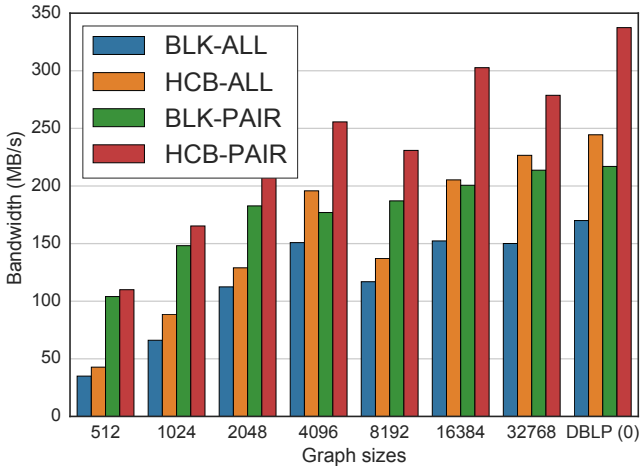


Figure 11: GSANA, Experiments on DBLP graphs on HW (8 nodelets).

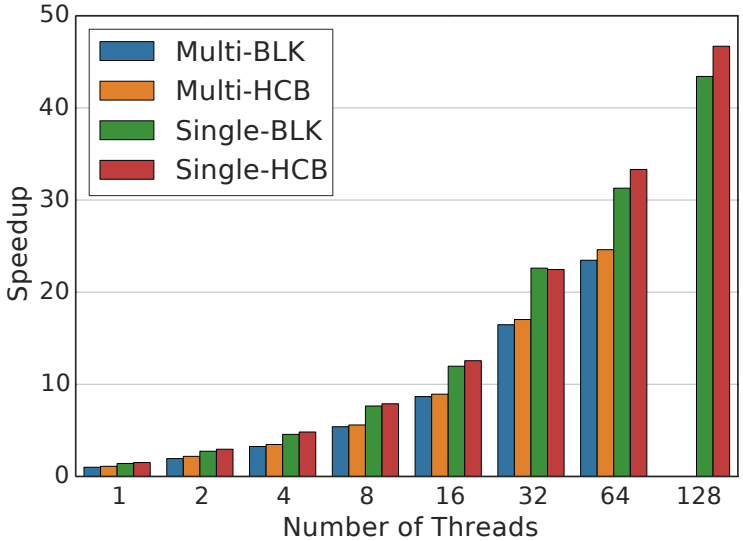


Figure 12: GSANA, strong scaling experiments on DBLP graph (2048 vertices) on HW (Multi-node and single-node).

6 Emu Architectural Discussion

The Emu architecture inverts the traditional scheme of hauling data to and from a grid of processing elements. In this architecture, the data is static, and small logical units of computation move throughout the system, and the load balancing is closely related to data layout and distribution, since threads can only run on local processing elements. During development, we encountered surprises that sometimes imposed $\geq 10\times$ execution time penalties. Our work mapping irregular algorithms to the Emu architecture expose the following issues needed to achieve relatively high performance:

1) **Thread stack placement** and remote thread migrations back to a "home" nodelet that contains the thread stack.
2) **Balancing the workload** is difficult when using irregular data structures like graphs.
3) The Emu is a **non-uniform PGAS** system with variable costs for remote "put" and "get" operations.
4) The tension between **top-down task programming on bottom-up data allocation** has proven difficult to capture in current programming systems.

Thread stack placement: A stack frame is allocated on a nodelet when a new thread is spawned. Threads carry their registers with them when they migrate, but stack accesses require a migration back to the originating nodelet. If a thread needs to access its stack while manipulating remote data, it will migrate back and forth (ping-pong). We can limit the usage of thread stacks and ping-pong migration by obeying the following rules when writing a function that is expected to migrate:

1) Maximize the use of inlined function calls. Normal function calls require a migration back to the home nodelet to save the register set.
2) Write lightweight worker functions using fewer than 16 registers to prevent stack spills.
3) Don't pass arguments by reference to the worker function. Dereferencing a pointer to a variable inside the caller's stack frame forces a migration back to the home nodelet. Pointers to replicated data circumvent this migration.

Workload balance and distributed data structures: One of the main challenges in obtaining good performance on the Emu Chick prototype is the initial placement of data and distribution to remote nodelets. While the current Emu hardware contains a hardware-supported credit system to control the overall amount of dynamic parallelism the choice of placement is still critical to avoid thread migration hotspots for SpMV and BFS. In the case of SpMV, replication reduces thread migration in each iteration, but replication is also not scalable to more complex, related algorithms like MTTKRP or SpGEMM. The implementations of graph alignment using gsaNA uses data placement techniques like HCB and PAIR-wise comparisons to group threads on the same nodelets with related data and limit thread migration, which dramatically improves their performance.

Non-uniform PGAS operations: Emu's implementation of PGAS utilizes "put"-style remote operations (add, min, max, etc.) and "get" operations where a thread is migrated to read a local piece of data. Thread migration is efficient when many get operations need to access the same nodelet-local memory channel. The performance difference observed between put and get operations is due to how these two operations

interact differently with load balancing. A put can be done without changing the location of the thread, while a get means that multiple threads may have to share significant resources on the same nodelet for a while. Additionally, a stream of gets with spatial locality can be faster than multiple put operations. This non-uniformity means that kernels that need to access finely grained data in random order should be implemented as put operations wherever possible while get operations should only be used when larger chunks of data are read together. A major outstanding question is how this scheme compares with explicitly remote references plus task migrations via remote calls as in UPC++[5]. The trade-off between hardware simplicity and software flexibility is difficult to measure without implementations of both. Tractable abstract models miss implementation details like switch fabric traffic contention or task-switching cache overhead.

Top-down task programming on bottom-up data allocation: The Cilk-based fork/join model emphasizes a top-down approach to maximize parallelism without regard to data or thread location. Memory allocation on the Emu system, however, follows the bottom-up approach of UPC[19] or SHMEM[14]. The Cilk model allows quickly writing highly parallel codes, but achieving high performance (bandwidth utilization) requires controlling thread locations. We do not yet have a good way to relieve these tensions. Languages like Chapel[13] and X10[15] provide a high-level view of data distribution but lack implicit thread migration. The GAANA results on the highly dynamic variant in Algorithm 5 demonstrate how migrations on locality-emphasizing data distribution can achieve relatively high performance. To our knowledge there is little work on programming systems that incorporate *implicit and light-weight* thread migration, but Charm++[1] and Legion[7] provide experience in programming heavier-weight task migration and locality in different environments.

Note that the Emu compiler is rapidly evolving to include intra-node `cilk_for` and Cilk+ reducers. Experimental support became available at the time of writing and still is being evaluated. Balancing remote memory operations and thread migrations in reducer and parallel scan implementations for the Emu architecture is ongoing work.

7 Related Work

Advances in memory and integration technologies provide opportunities for profitably moving computation closer to data [62]. Some proposed architectures return to the older processor-in-memory (PIM) and "intelligent RAM" [56] ideas. Simulations of architectures focusing on near-data processing [32] including in-memory [31] and near-memory [30] show great promise for increasing performance while also drastically reducing energy usage.

Other hardware architectures have tackled massive-scale data analysis to differing degrees of success. The Tera MTA / Cray XMT[51, 24] provide high bandwidth utilization by tolerating long memory latencies in applications that can produce enough threads to source enough memory operations. In the XMT all memory interactions are remote incurring the full network latency on each access. The Chick instead moves threads to memory on reads, assuming there will be a cluster of reads for nearby data. The Chick processor needs to tolerate less latency and need not keep as many threads in flight. Also, unlike the XMT, the Chick runs the operating system on the stationary

processors, currently PowerPC, so the Chick processors need not deal with I/O interrupts and highly sequential OS code. Similarly to the XMT, programming the Chick requires language and library extensions. Future work with performance portability frameworks like Kokkos[25] will explore how much must be exposed to programmers. Another approach is to push memory-centric aspects to an accelerator like Sparc M7's data analytics accelerator[2] for database operations or Graphicionado[33] for graph analysis.

Moving computation to data via software has a successful history in supercomputing via Charm++[1], which manages dynamic load balancing on distributed memory systems by migrating the computational objects. Similarly, data analysis systems like Hadoop moved computation to data when the network was the primary data bottleneck [4]. The Emu Chick also is strongly related to other PGAS approaches and is a continuation of the mNUMA architecture [65]. Other approaches to hardware PGAS acceleration include advanced RDMA networks with embedded address translation and atomic operations [29, 58, 61, 21]. The Emu architecture supports remote memory operations to varying degrees and side-steps many other issues through thread migration. Remote operations pin a thread so that the acknowledgment can be delivered. How to trade between migration and remote operations, as well as exposing that trade-off, is an open question.

SpMV: There has been a large body of work on SpMV including on emerging architectures [67, 12] but somewhat limited recent work that is directly related to PGAS systems. However, future work with SpMV on Emu will investigate new state-of-the-art formats and algorithms such as SparseX, which uses the Compressed Sparse eXtended (CSX) as an alternative data layout for storing matrices [27].

BFS: The implemented version of BFS builds on the standard Graph500 code with optimizations for Cilk and Emu. The two-phase implementation used in this work has some similarities to direction-optimizing BFS [9], in that the remote "put" phase mirrors the bottom-up algorithm. Other notable current implementations include optimized, distributed versions [64] and a recent PGAS version [18]. The implementation provided in this paper contrasts with previous PGAS work due to asymmetric costs for remote get operations as discussed in Section 6. NUMA optimizations[70] similarly are read-oriented but lack thread migration.

Graph Alignment: Graph alignment methods are traditionally [20, 28] classified into four basic categories: spectral methods [63, 44, 55, 52, 74]; graph structure similarity methods [42, 49, 48, 46, 3]; tree search or table search methods [17, 60, 45, 40]; and integer linear programming methods [36, 26, 8, 39]. `Final` [74] is a recent work which targets labeled network alignment problem by extending the concept of `IsoRank` [63] to use attribute information of the vertices and edges. All of these methods have scalability issues. GSANA [68, 69] leverages global graph structure and reduces the problem space and exploits the semantic information to alleviate most of the scalability issues. In addition to these sequential algorithms, we are aware of two parallel approaches for global graph alignment. The first one [37] decomposes the ranking calculations of `IsoRank`'s similarity matrix using the singular value decomposition. The second one is a shared memory parallel algorithm [53] that is based on the belief propagation (BP) solution for integer program relaxation [8]. It uses

shared memory parallel matrix operations for BP iterations and also implements an approximate weighted bipartite matching algorithm. While these parallel algorithms show an important improvement over the state of the art sequential algorithms, the graphs used in the experiments are small in size and there is a high structural similarity. To the best of our knowledge, the use of GSANA in [69] and in this paper presents the first method for parallel alignment of labeled graphs.

Other recent work has also looked to extend from low-level characterizations like those presented here by providing initial Emu-focused implementations of Breadth-First Search[11], Jaccard index computation [41], bitonic sort, [66] and compiler optimizations like loop fusion, edge flipping, and remote updates to reduce migrations [16].

8 Conclusion

In this study, we focus on optimizing several irregular algorithms using programming strategies for the Emu system including replication, remote writes, and data layout and placement. We argue that these three types of programming optimizations are key for achieving good workload balance on the Emu system and that they may even be useful to optimize Cilk-oriented codes for x86 systems (as with BFS).

By analogy, back-porting GPU-centric optimizations to processors often provides improved performance. That is, in the same way that GPU architecture and programming encourages (or "forces") programmers to parallelize and vectorize explicitly, the Emu design requires upfront decisions about data placement and one-sided communication that can lead to more scalable code. Future work would aim to evaluate whether these programming strategies can be generalized in this fashion.

By adopting a "put-only" strategy, our BFS implementation achieves 80 MTEPS on balanced graphs. Our SpMV implementation makes use of replicated arrays to reach 50% of measured STREAM bandwidth while processing sparse data. We present two parallelization schemes and two vertex layouts for parallel similarity computation with the GSANA graph aligner, achieving strong scaling up to $68\times$ on the Emu system. Using the *HCB* vertex layout further improves the execution time by up to 36%.

Acknowledgments

This work partially was supported by NSF Grant ACI-1339745 (XScala), an IARPA contract, and the Defense Advanced Research Projects Agency (DARPA) under agreement #HR0011-13-2-0001. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors. Thanks also to the Emu Technology team for support and debugging assistance with the Emu Chick prototype, and to the reviewers for their thoughtful comments.

References

[1] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale. Parallel programming with migratable objects: Charm++ in practice. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 647–658, November 2014.

[2] K. Aingaran, S. Jairath, G. Konstadinidis, S. Leung, P. Loewenstein, C. McAllister, S. Phillips, Z. Radovic, R. Sivaramakrishnan, D. Smentek, and T. Wicki. M7: Oracle's next-generation SPARC processor. *IEEE Micro*, 35(2):36–45, March 2015.

[3] Ahmet E Aladağ and Cesim Erten. Spinal: scalable protein interaction network alignment. *Bioinformatics*, 29(7):917–924, 2013.

[4] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Disk-locality in datacenter computing considered irrelevant. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[5] J. Bachan, S. Baden, D. Bonachea, P. Hargrove, S. Hofmeyr, M. Jacquelin, A. Kamil, and B. van Straalen. Upc++ specification v1.0, draft 8. 2018.

[6] David A Bader, Jonathan Berry, Simon Kahan, Richard Murphy, E Jason Riedy, and Jeremiah Willcock. Graph500 benchmark 1 (search) version 1.2. Technical report, Graph500 Steering Committee, September 2011.

[7] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[8] Mohsen Bayati, Margot Gerritsen, David F Gleich, Amin Saberi, and Ying Wang. Algorithms for large, sparse network alignment problems. In *IEEE International Conference on Data Mining (ICDM)*, pages 705–710, 2009.

[9] S. Beamer, A. Buluç, K. Asanovic, and D. Patterson. Distributed memory breadth-first search revisited: Enabling bottom-up search. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pages 1618–1627, May 2013.

[10] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[11] Mehmet Belviranli, Seyong Lee, and Jeffrey S. Vetter. Designing algorithms for the EMU migrating-threads-based architecture. *High Performance Extreme Computing Conference 2018*, 2018.

[12] Dan Bonachea, Rajesh Nishtala, Paul Hargrove, and Katherine Yelick. Efficient point-to-point synchronization in upc. In *2nd Conf. on Partitioned Global Address Space Programming Models (PGAS06)*, 2006.

[13] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.

[14] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 2:1–2:3, New York, NY, USA, 2010. ACM.

[15] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[16] Prasanth Chatarasi and Vivek Sarkar. A preliminary study of compiler transformations for graph applications on the Emu system. In *Proceedings of the Workshop on Memory Centric High Performance Computing*, MCHPC'18, pages 37–44, New York, NY, USA, 2018. ACM.

[17] Leonid Chindelevitch, Cheng-Yu Ma, Chung-Shou Liao, and Bonnie Berger. Optimizing a global alignment of protein interaction networks. *Bioinformatics*, 29(21):2765–2773, 09 2013.

[18] Guojing Cong, George Almasi, and Vijay Saraswat. Fast pgas implementation of distributed graph algorithms. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[19] UPC Consortium, Dan Bonachea, and Gary Funck. Upc language and library specifications, version 1.3. 11 2013.

[20] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty years of graph matching in pattern recognition. *International journal of pattern recognition and artificial intelligence*, 18(03):265–298, 2004.

[21] Michael Dungworth, James Harrell, Michael Levine, Stephen Nelson, Steven Oberlin, and Steven P. Reinhardt. *CRAY T3E*, pages 419–441. Springer US, Boston, MA, 2011.

[22] Timothy Dysart, Peter Kogge, Martin Deneroff, Eric Bovell, Preston Briggs, Jay Brockman, Kenneth Jacobsen, Yujen Juan, Shannon Kuntz, and Richard Lethin. Highly scalable near memory processing with migrating threads on the Emu system architecture. In *Irregular Applications: Architecture and Algorithms (IA3), Workshop on*, pages 2–9. IEEE, 2016.

[23] David Ediger, Robert McColl, Jason Riedy, and David A. Bader. STINGER: High performance data structure for streaming graphs. In *The IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, September 2012.

[24] David Ediger, Jason Riedy, David A. Bader, and Henning Meyerhenke. Computational graph analytics for massive streaming data. In Hamid Sarbazi-azad and Albert Zomaya, editors, *Large Scale Network-Centric Computing Systems*, Parallel and Distributed Computing, chapter 25. Wiley, July 2013.

[25] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[26] Mohammed El-Kebir, Jaap Heringa, and Gunnar W Klau. Lagrangian relaxation applied to sparse global network alignment. In *IAPR International Conference on Pattern Recognition in Bioinformatics*, pages 225–236. Springer, 2011.

[27] Athena Elafrou, Vasileios Karakasis, Theodoros Gkountouvas, Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. SparseX: A library for high-performance sparse matrix-vector multiplication on multicore platforms. *ACM Trans. Math. Softw.*, 44(3):26:1–26:32, January 2018.

[28] Ahed Elmsallati, Connor Clark, and Jugal Kalita. Global alignment of protein-protein interaction networks: A survey. *IEEE Transactions on Computational Biology and Bioinformatics*, 13(4):689–705, 2016.

[29] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard. Cray cascade: A scalable hpc system based on a dragonfly network. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–9, November 2012.

[30] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim. Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 283–295, February 2015.

[31] T. Finkbeiner, G. Hush, T. Larsen, P. Lea, J. Leidel, and T. Manning. In-memory intelligence. *IEEE Micro*, 37(4):30–38, August 2017.

[32] M. Gao, G. Ayers, and C. Kozyrakis. Practical near-data processing for in-memory analytics frameworks. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 113–124, October 2015.

[33] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, October 2016.

[34] Eric Hein and Thomas M. Conte. Dynograph: Benchmarking dynamic graph analytics. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2016. Poster.

[35] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluc, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, and et al. Mathematical foundations of the graphblas. *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep 2016.

[36] Gunnar W. Klau. A new graph-based method for pairwise global network alignment. *BMC Bioinformatics*, 10(1):S59, 2009.

[37] Giorgos Kollias, Shahin Mohammadi, and Ananth Grama. Network similarity decomposition (nsd): A fast and scalable approach to network alignment. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 24(12):2232–2243, 2012.

[38] Scott Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. The SuiteSparse matrix collection website interface. *Journal of Open Source Software*, 4(35):1244, Mar 2019.

[39] Danai Koutra, Hanghang Tong, and David Lubensky. Big-align: Fast bipartite graph alignment. In *IEEE International Conference on Data Mining (ICDM)*, pages 389–398, 2013.

[40] Segla Kpodjedo, Philippe Galinier, and Giulio Antoniol. Using local similarity measures to efficiently address approximate graph matching. *Discrete Applied Mathematics*, 164:161 – 177, 2014. Combinatorial Optimization.

[41] Ǵeraud P. Krawezik, Peter M. Kogge, Timothy J. Dysart, Shannon K. Kuntz, and Janice O. McMahon. Implementing the Jaccard index on the migratory memory-side processing Emu architecture. *High Performance Extreme Computing Conference 2018*, 2018.

[42] Oleksii Kuchaiev, Tijana Milenković, Vesna Memišević, Wayne Hayes, and Nataša Pržulj. Topological network alignment uncovers biological function and phylogeny. *Journal of the Royal Society Interface*, 7(50):1341–1354, 2010.

[43] Charles E Leiserson. Programming irregular parallel applications in Cilk. In *International Symposium on Solving Irregularly Structured Problems in Parallel*, pages 61–71. Springer, 1997.

[44] Chung-Shou Liao, Kanghao Lu, Michael Baym, Rohit Singh, and Bonnie Berger. Isorankn: spectral methods for global alignment of multiple protein networks. *Bioinformatics*, 25(12):i253–i258, 2009.

[45] D. Liu, K. C. Tan, C. K. Goh, and W. K. Ho. A multiobjective memetic algorithm based on particle swarm optimization. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 37(1):42–50, 2007.

[46] Noël Malod-Dognin and Nataša Pržulj. L-GRAAL: Lagrangian graphlet-based network aligner. *Bioinformatics*, 31(13):2182–2189, 02 2015.

[47] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[48] Vesna Memišević and Nataša Pržulj. C-graal: Common-neighbors-based global gra ph al ignment of biological networks. *Integrative Biology*, 4(7):734–743, 2012.

[49] T. Milenković, W. Ng, W. Hayes, and Natasa Przulj. Optimal network alignment with graphlet degree vectors. *Cancer Informatics*, 9:121 – 137, 2010.

[50] Marco Minutoli, S Kuntz, Antonino Tumeo, and P Kogge. Implementing radix sort on Emu 1. In *3rd Workshop on Near-Data Processing (WoNDP)*, 2015.

[51] D. Mizell and K. Maschhoff. Early experiences with large-scale cray xmt systems. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–9, May 2009.

[52] Behnam Neyshabur, Ahmadreza Khadem, Somaye Hashemifar, and Seyed Shahriar Arab. Netal: a new graph-based method for global alignment of protein–protein interaction networks. *Bioinformatics*, 29(13):1654–1662, 2013.

[53] Behnam Neyshabur, Ahmadreza Khadem, Somaye Hashemifar, and Seyed Shahriar Arab. Netal: a new graph-based method for global alignment of protein–protein interaction networks. *Bioinformatics*, 29(13):1654–1662, 2013.

[54] University of Trier. DBLP: Computer science bibliography. `http://dblp.dagstuhl.de/xml/release/`, 2017.

[55] Rob Patro and Carl Kingsford. Global network alignment using multiscale spectral signatures. *Bioinformatics*, 28(23):3105–3114, 2012.

[56] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent RAM. *IEEE Micro*, 17(2):34–44, March 1997.

[57] R. Pearce, M. Gokhale, and N. M. Amato. Faster parallel traversal of scale free graphs at extreme scale with vertex delegates. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 549–559, November 2014.

[58] R. Rajamony, L. B. Arimilli, and K. Gildea. Percs: The ibm power7-ih high-performance computing system. *IBM Journal of Research and Development*, 55(3):3:1–3:12, May 2011.

[59] Thomas B. Rolinger and Christopher D. Krieger. Impact of traditional sparse optimizations on a migratory thread architecture. *2018 IEEE/ACM 8th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, Nov 2018.

[60] Vikram Saraph and Tijana Milenković. Magna: maximizing accuracy in global network alignment. *Bioinformatics*, 30(20):2931–2940, 2014.

[61] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. Stunkel, G. Bosilca, and A. Bouteiller. Ucx: An open source framework for hpc network apis and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 40–43, August 2015.

[62] Patrick Siegl, Rainer Buchty, and Mladen Berekovic. Data-centric computing frontiers: A survey on processing-in-memory. In *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, pages 295–308, New York, NY, USA, 2016. ACM.

[63] Rohit Singh, Jinbo Xu, and Bonnie Berger. Pairwise global alignment of protein interaction networks by matching neighborhood topology. In *Annual International Conference on Research in Computational Molecular Biology*, pages 16–31, 2007.

[64] Koji Ueno, Toyotaro Suzumura, Naoya Maruyama, Katsuki Fujisawa, and Satoshi Matsuoka. Efficient breadth-first search on massively parallel and distributed-memory machines. *Data Science and Engineering*, 2(1):22–35, March 2017.

[65] Megan Vance and Peter M. Kogge. Introducing mnuma: An extended pgas architecture. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 6:1–6:10, New York, NY, USA, 2010. ACM.

[66] Kaushik Velusamy, Thomas B. Rolinger, Janice McMahon, and Tyler A. Simon. Exploring parallel bitonic sort on a migratory thread architecture. *High Performance Extreme Computing Conference 2018*, 2018.

[67] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix–vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, 2009.

[68] Abdurrahman Yaşar and Ümit V. Çatalyürek. An iterative global structure-assisted labeled network aligner. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Aug 2018.

[69] Abdurrahman Yaşar, Bora Uçar, and Ümit V. Çatalyürek. Sina: A scalable iterative network aligner. In *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, 2018.

[70] Yuichiro Yasui, Katsuki Fujisawa, and Yukinori Sato. Fast and energy-efficient breadth-first search on a single numa system. In Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer, editors, *Supercomputing*, pages 365–381, Cham, 2014. Springer International Publishing.

[71] Chunxing Yin, Jason Riedy, and David A. Bader. A new algorithmic model for graph analysis of streaming data. In *Proceedings of the 14th International Workshop on Mining and Learning with Graphs (MLG)*, May 2018.

[72] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 25–25, November 2005.

[73] Jeffrey S. Young, Eric Hein, Srinivas Eswar, Patrick Lavin, Jiajia Li, Jason Riedy, Richard Vuduc, and Thomas M. Conte. A microbenchmark characterization of the Emu Chick. *Parallel Computing*, 87:60–69, Sep 2019.

[74] Si Zhang and Hanghang Tong. Final: Fast attributed network alignment. In *ACM International Conference on Knowledge Discovery and Data mining*, pages 1345–1354, 2016.